On the importance of Java to ATLAS

Steve Fisher

<s.m.fisher@rl.ac.uk>

Julius Hrivnac

<julius.hrivnac@cern.ch>

15th November 2000

ATL-SOFT-2000-004

Dec 2000

Table of Contents

The Java programming language has had a huge impact in the world outside HEP and is now making itself noticed within HEP. Here we discuss why we should consider it, and what the impact might be.

1. Introduction

If HEP were to start the migration from FORTRAN to OO today, it is very likely that the choice would be Java.

This does not imply any criticism of the original choice of C++. At the time it seemed sensible; Java had only just appeared as something for building web toys. The world has moved on. The arguments that previously favoured C++ now point to Java. Nobody would promote C++ as a safe or easy language, yet Java exhibits both characteristics.

Java appeared about five years ago and is used in a number of recently developed applications. For example most XML tools are Java based. In HEP it has been demonstrated mainly in highly graphical applications such as JAS (Java Analysis Studio) or the WIRED Event Display, but it is also being used for the reconstruction software (hep.lcd) being developed in the LCD collaboration at SLAC.

The main part of this document is made up of lists of Java features which have been categorised as follows:

Section 2.	lists those features of Java which make it clearly better than C++
Section 3.	also a pro-Java category which explains those features which were handled badly when the language was young, but have now been solved
Section 4.	short list of possible problems
Section 5.	short list of unresolved problems.
Section 6.	lists those features which are neither good nor bad; they are things which Java just does differently.

Finally Section 7. summarises the potential of Java in HEP and Section 8. makes specific recommendations to ATLAS concerning the future Java strategy.

2. Benefits of Java

One of our collaborators was heard to say "C++ makes writing code much more of a challenge." This is almost certainly true and is the reason why we should move away from C++ - it is needlessly difficult to get right. People should be able to concentrate on the algorithms and not worry about the details of memory management. We want to empower people to be able to contribute towards the ATLAS software effort and so we cannot afford the complexity of C++. It is remarkably easy to become fluent in Java, you may need to look up the details of some class you wish to use but rarely need to check in a textbook for details of the language itself.

2.1 Automatic memory management

There is no need to delete objects manually with Java. Like most languages (apart from C++) objects get deleted when they are no longer reachable. This does not completely free the programmer from the burden of thinking about who owns objects as it is still possible to prevent objects being garbage collected by keeping a reference somewhere. However it is not possible to follow a pointer to some piece of memory where an object used to be. If you set a reference to null, and try to use it you get an exception immediately.

2.2 No "*", "->" or "&" operators to worry about just "."

act as C++ pointers to objects but you use "." instead of "->". If you want to copy an object you must explicitly clone it, as assignment just sets a reference to what is on the right hand side of the assignment.

2.3 Clean definition of interfaces

Java encourages the programmer to think about interfaces. An interface specifies the set of methods which characterise it. A class may implement many interfaces. Note that by contrast a class may only extend (i.e. inherit from) one class; this avoids the complex C++ rules about what happens when a class appears twice in the inheritance tree.

2.4 No ".h" files

Keeping headers and code consistent in C++ can be quite painful. In Java there are are no headers and so instead of #including a header file, the compiler is able to obtain the information it needs from the other Java files or from the already compiled files. It is true that .h files make good documentation (except that they also show the private member data) but the information which can be extracted by the Javadoc program, which is a part of the standard Java distribution, is much better.

2.5 World-wide naming scheme

Java recommends a package naming scheme which if followed guarantees the world wide uniqueness of every class name. This should mean that you can use any library without fear of name clashes.

2.6 No preprocessor

Multi-platform development is very easy because the Java source code will compile on multiple platforms and the resulting byte code will then execute anywhere - this is even true between Linux and NT. C++ has the preprocessor which can handle conditional code (e.g. #ifdef) to cope with platform variations. For Java there is no preprocessor and so no conditional code. This makes program testing much simpler.

2.7 Byte code can execute anywhere with a JVM

A Java compiler does not produce machine code but portable byte code. The Java Virtual Machine(JVM) is the software which executes the portable Java byte code.

Java is evolving so you should use a recent JVM. You would want to do this anyway to benefit from the performance these JVMs offer. One of the more exciting ways to exploit this feature is with mobile agents. Code compiled on NT really runs properly under Linux and vice versa.

2.8 Fast development

The compilers are fast and give clear diagnostics and there is no linking step to be carried out. This results in a very fast program development cycle and faster prototyping.

Code is loaded on demand, so you pay no penalty for executing only a small part of a big program. This is similar to dynamic linking in other programming languages, but with Java it was designed in from the beginning - so it works effortlessly and reliably

2.9 Fast execution

With the latest technology from IBM and SUN , as described in Section 3.1, Java programs have performance comparable to their C++ equivalent.

2.10 Thread management is part of the language

This may seem to be of limited interest to most of us in the offline. However there are real uses - for example to avoid delays waiting for the next event to be read. For a GUI to behave in a reasonable way threads are essential; otherwise the GUI is blocked while the application part

2.11 Exact reproducibility of results

Currently reproducibility of results across hardware is part of the design of the language. For reasons of efficiency it is now possible to turn this off. However the benefit of expecting bit for bit identical results on different hardware should not be underestimated. We would need to check in our applications, if and when it is worth sacrificing reproducibility of results across hardware, in favour of efficiency.

2.12 "Reflection" facilities

Reflection is the Java term for the ability of a program to learn all about the objects it contains. This allows you to do tricky things - but still in a type-safe manner. It makes interfaces to databases very elegant and is the underlying support for a lot of the features which are taken for granted with Java - for example excellent (and portable) debuggers, serialization (see Section 2.13 and JavaBeans.

2.13 "Serialization"

Any class which implements the `Serializable` interface can be serialized, i.e. written out to to a file. You just have to put `implements Serializable` after the class name in the declaration of the class and it can be output with no effort.

2.14 Dynamic optimisation

A huge amount of effort has been invested in various technologies to make Java execution fast. This work has resulted in claims of better performance for Java than C++. Most of the work centres around efficient garbage collection and upon dynamic optimisation. The virtual machine notices where the code is spending its time and optimises that section. A normal JIT compiler converts byte code to machine code the first time it is executed. The newer technologies profile the code as it is executing and decide which parts are worth the cost of converting to optimised machine code.

2.15 Dynamic loading

There is no huge executable to build; instead classes are loaded dynamically as they are needed. This also gives extra flexibility in complex systems as the application can decide itself which libraries to use. You can even write your own class loader if you have unusual requirements.

2.16 Javadoc generated documentation

By simply following the Javadoc conventions as recommended in ATL-SOFT-2000-002 http://weblib.cern.ch/format/showfull?base=ATLATL&sysnb=0001741 excellent package documentation can be generated automatically.

2.17 Excellent tool support

Tool support for Java is excellent with many free and commercial tools available. For example the CASE tool, Together, supports both C++ and Java but new features (such as audits, metrics and patterns) are always released first for Java.

2.18 Easy to configure for compilation

To compile Java code you only need to know where the source file (or files) are, and the locations of the already compiled files. The directories with compiled files (or a compressed archive) are normally specified with the `CLASSPATH` environment variable. The various compilers and IDE tools use this, and small programs have no need for make or similar programs to control the build sequence.

3. Non-problems

are not problems at all or are problems in any programming language because they reflect the difficulty of the problem they are solving.

3.1 Performance for numerical operations

A frequent complaint heard about Java is its speed. This often comes from people who just assume "java = interpreted = slow", or from people whose primary experience of Java is waiting for applets to download into Netscape. It has been demonstrated several times, as indicated in the references just below, that intensive numerical calculations in Java are of comparable performance to those in C++. The code which actually runs for a Java program is real compiled code generated by a JIT or HotSpot compiler. The performance dependency on benchmark parameters indicates that the actual executable code is very similar for Java and C++; the same patterns connected to e.g. cache size is observed in both Java and C++ programs. The performance depends much more on the choice of algorithm, libraries and program design than on the language. New HotSpot compilers allow behaviour-driven dynamic compilation and profiling of the compiled Java code. While this may make short programs slower due to the profiling overhead, longer programs can be optimized to a level inaccessible to standard techniques used for C++ optimization. Java has already reached the performance of the standard optimized C/C++ programs and it approaches the performance of programs designed for performance in C/C++/F77.

IBM's Java performance research	http://www.research.ibm.com/journal/sj39-1.html
C vs. Java speed comparisons	http://www.aceshardware.com/Spades/read.php?article_id=153
Java versus C++	http://www.multimania.com/lefevre/java/ceaxx.html
Java Performance	http://hepunx.rl.ac.uk/hep2java/meeting/16-03-2000/hoschek2/sld001.htm
Java Performance	http://atlas.web.cern.ch/Atlas/GROUPS/GRAPHICS/Texts/Taylor-00.05.12/
Java Grande Forum	http://www.javagrande.org/
Java for Scientific Computing	http://www.npac.syr.edu/projects/javaforcse/
The Java HotSpot Performance Engine Architecture	http://java.sun.com/products/hotspot/whitepaper.html

3.2 3D graphics performance

The speed and quality of 3D graphics in Java applications *used to be* inferior to that of applications based on C/C++ with OpenGL/OpenInventor as the foundation libraries. Java3D and OpenInventor both use OpenGL/Mesa as the mechanism for the actual 3D drawing so can both profit to the same extent from the hardware acceleration available from modern PC video-cards. Coupled with the performance improvements of the JVM, applications built on top of Java3D now have similar performance to those built in C/C++ with OpenInventor. The final performance depends primarily on the design and implementation of the visualization libraries and programs.

Java3D http://java.sun.com/products/java-media/3D/index.html

3.3 Numerics

Java is designed to give exactly reproducible results (including floating point operations) on all platforms, all compilers and all Virtual Machines. This feature is very important when results from variety of platforms should be combined. However, there are certain tradeoffs connected to it as hardware features available only on some platforms can't be always fully exploited. This may sometimes lead to the loss of speed and precision.

numbers. To guarantee exact reproducibility, Java rounds such number into standard Double (64bit) precision. This leads to the loss of precision and speed in performing the rounding operation. For the same reason, Java doesn't allow fused multiply-accumulate which slows some operations such as matrix manipulations. The exact reproducibility has also certain optimization consequences: for example the compiler is not free to reorder associative operations.

The problem of not being able to fully exploit the hardware has been mostly *solved by Java 1.3*, which can internally used float extended-exponent and double extended-exponent. A new keyword, strictfp, has been introduced to enforce exact reproducibility (which is now *not* the default). Java also contains two mathematical libraries, java.lang.Math (for precise and fast mathematics) and java.lang.StrictMath (for exactly reproducible mathematics). Any remaining performance problems connected with the exact reproducibility have been largely outweighed by the superb numerics built in to the new JDK 1.3 from IBM.

Java doesn't allow access to the rounding mode and only does "round to nearest" (IEEE 754 default). However in C++, rounding mode is implementation specific. Access to all IEEE 754 modes may be available, but portable code can't rely on it.

Java doesn't have flags for IEEE 754 Exceptions: Invalid Operation, Overflow, Division-by-Zero, Underflow and Inexact Result. So the program can't handle them easily. Instead the program takes some default action which may not be always optimal. Java only defines one NaN (not-a-number), while IEEE 754 specifies several. C++ may turn floating point exceptions into C++ exceptions, but again it is implementation specific.

In summary the numeric problems have either been solved in the current release of the Java specification, or the C++ situation is no better than the Java one. C++ implementations may give wider control over the processing hardware, but Java always allows the user to choose between exactly reproducible results or to use the hardware to optimise speed and precision.

How Java's Floating-Point Hurts Everyone Everywhere	http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf
Updates to the Java Language Specification for JDK Release 1.2 Floating Point	http://java.sun.com/docs/books/jls/strictfp-changes.pdf

3.4 Memory Management

Automatic garbage collection is another strong point of Java. The limited possibility to control it is often mentioned as a problem for larger Java programs. Memory management can be optimized in C++ but there is no control in Java. Java doesn't allow the developer to cluster objects efficiently in memory.

Sun's HotSpot compiler contains advanced memory allocation and garbage collection mechanisms, which is in most cases more effective than anything done by hand by a knowledgeable C++ programmer or by an add-on C++ garbage collector. If a user really needs to control when the collection happens, he or she can trigger it so as to reduce the likelihood of it having a major impact at the wrong moment in an interactive programme. This possibility is, however, very rarely needed as the standard HotSpot garbage collector collects and re-clusters memory fluently. Weak references offer another way of efficiently interacting with the object management system and allows implementation of sophisticated ways of object caching.

The Java HotSpot Performance Engine Architecture	http://java.sun.com/products/hotspot/whitepaper.html

4. Possible problems

These are problems which may be concealed in not-yet fully tested features or in application domains where Java has not yet been tried.

4.1 Handling of too many objects

HotSpot memory manager performs especially well in those cases and is able to prevent memory fragmentation.

4.2 Running in the Event Filter and Level 2 Trigger

Java should be tested in the Event Filter and Level 2 Trigger to ensure that it meets the performance requirements. No significant problems connected to the performance or memory management are expected.

5. Unresolved problems

These are real problems for which a solution should be found.

5.1 Access to databases

Java can either access objects in a database based on C++ (where C++ objects should be converted to the Java objects on-the-fly) or directly in a Java database. While there are candidates for a suitable database using C++ technology, experience with Java-based databases is much smaller. Some promising candidates (see below) for a fully-featured Java-based OO database suitable for HEP have appeared which seem to be quite close to satisfying our functional requirements but performance and scalability are not known.

The experience from other areas, such as CASE tools, shows many small-scale applications appearing which reflects a real need for that application. This is shortly followed by mature usable applications. This is exactly what is happening today with Java databases: the range of candidate OO databases is very wide from the simple Open Software products such as Ozone, through the Java binding of the traditional OO databases such as Objectivity to the new Java-specific OO database solutions such as JDO and Forest.

Objectivity Java Binding	http://www.objy.com/Products/Java/Objy5JAVA.htm
Ozone	http://www.ozone-db.org/
JDO	http://java.sun.com/aboutJava/communityprocess/review/jsr012/JDO_0_8.pdf
Forest	http://www.sun.com/research/forest/

5.2 Java <-> C++ collaboration

It is generally difficult to interface different OO languages. As some HEP code has already been written in C++ and more code may be written, the question of interfacing between C++ and Java is frequently seen as the major obstacle for introducing Java into the HEP environment on a wide scale. While this argument may be used to advocate the most rapid possible migration to Java, it is still worth considering the Java <-> C++ inter-operation to make use of significant C++ packages for example.

Various possible language inter-operation scenarios have been investigated as indicated in the reference below. The most advanced seems to be the JACO project, aiming at the transparent access of Java to C++ (and vice versa). Experience accumulated so far in HEP indicates that the language inter-operation could be automated provided that strict interfaces are defined between those packages which must communicate and are written in different languages. Having a well defined simple interface is good practice, so having packages in different languages may be of some benefit as it forces us to define interfaces clearly.

Meeting on HEP migration to Java - C++/Java interworking	http://hepunx.rl.ac.uk/hep2java/meeting/25-05-2000/

5.3 Non-existence of immutable objects

There is no equivalent of the C++ `const` keyword in Java. There is only `final`, which is very different from `const` and can rarely be used as an equivalent. In particular there is no keyword to mark formal parameters as immutable.

that this problem will disappear once people become accustomed to the fact that they are dealing with *the object* rather than a copy of it and will then treat it with care. Further protection can be offered by ensuring that any "set" methods only have package visibility so a user from another package cannot do any harm.

Another technique is to define "const interfaces" for objects which are not to be modified. The idea of a const interface, which a class can implement, is that it only exposes methods which return simple data and const interfaces to other objects. A const interface is not generic but must be defined for each class; for example `ConstEvent`, `ConstTrack` and `ConstHit`. Note that these are *not* extra objects but just safe interfaces to existing objects.

6. Other features

Those are just the features of the language. They may sometimes be seen as benefits or problems.

No multiple (non-abstract) inheritance. This is very often the sign of a good OO style even in other OO languages.

No operator overloading. For non-primitive types, operator can't be overloaded. For example, vectors can't be added as `A + B`, but only as `A.plus(B)`. There are attractions in allowing operator overloading to support mathematical operations where the notation is close to the normal mathematical formalism and one can take advantage of the operator precedence rules.

No automatic type narrowing. The type conversion which could lead to a loss of precision doesn't happen automatically, but variables must be cast explicitly. This protects the developer from certain hard-to-find bugs and makes the behaviour of the code explicit, but some may find it annoying.

No genericity. Unrestricted genericity, where the type is an argument to a class to be used when the object is instantiated, as implemented for example in C++ as templates, represents a programming paradigm largely orthogonal to OO. Very often, C++ templates are not used for generic programming but to solve other C++ problems (for example non-existence of reflection, so the programmer should supply the type information which could otherwise be extracted from the object itself). Compilation of C++ templates still poses serious problem for some compilers. In Java, most of the C++ templated constructs can be rewritten using reflection, `Class` and `Object` objects.

The place where templates are useful is in constructing typed collections. The standard Java collections will hold anything. It is only when you get the object and cast it to the type you expect that you discover it is not what you had anticipated. However it is easy to write a wrapper class which will look almost exactly like an STL container; you can constrain the type of object it will hold by passing an object or class to the constructor of the collection.

No macros. The only legitimate use of macros in C++ is to solve the platform dependency problem (Section 2.6). This means that macros are not needed in Java.

No global functions. They are not needed in the real OO language. Also static methods provide the same functionality without taking global name space.

No inlined functions. Even in C++, the compiler often does a better job than the programmer.

No `typedef`. The `typedef` in C++ is normally used to hide the complexities of generic collections. This is clearly not a problem for Java.

No `enum`. It would be nice to have enum. To get a similar effect you can define a series of `static final int`s but you have to look after the values yourself.

Arrays are only one dimensional. This is not too serious with the new highly optimized mathematical libraries from IBM. You can have arrays of arrays. This can be used to represent triangular matrices very directly.

Array bounds and null pointer checking. While these features cause certain performance degradation, it makes program execution much safer. Exceptions (from out-of-range indexes and null pointers) happen immediately and are easy to pinpoint. In C and C++, the checking

compilers can to a large extent remove the performance problem by analyzing the code and identifying which checks are superfluous.

String is 16bit Unicode (not 8bit ASCII). Strings are not used so often in HEP to create problems with space. The advantage of the Unicode is easy internationalization and easy access to non-Latin-1 characters (like Greek alphabet).

Java isn't (ISO) standard. The current Java Community Process works fine in introducing new useful features while granting reasonable language stability.

7. Place for Java in HEP

Java is emerging as the best general purpose computing language available at the moment and as such we expect it to become the 'default' language for most tasks in HEP.

7.1 Framework

Java's features (natural dynamic loading, platform independence, network awareness, reflection,...) make it ideal for the building of the Framework. The Framework should allow other languages as algorithms or services. We should be ready for other languages which will become important but which may not have been invented yet. Java is already being successfully used for the Framework (and all its components) in the work for the LCD experiment.

7.2 Reconstruction

As there's no performance problem in Java and the algorithmic code is almost identical in Java and C++, reconstruction algorithms can easily fit into the Java environment. Algorithms written in either language should be able to transparently collaborate. The "ordinary physicist" is especially involved in the reconstruction and simulation and even more so in the analysis. The benefits of coding in a robust language are immense in these areas.

7.3 Simulation

The current mainstream simulation strategy in HEP is based on the G4, which is written in C++ and it's unrealistic to expect it to be re-written in any other language soon. Also the key particle generators are written in various other languages (Fortran, C++) and some of them will stay as they are. For those reasons, it's clear, that if Java is massively used in HEP, Simulation should be reasonably interfaced. This should, however, not be very difficult as the interface between simulation and other domains is clearly defined.

7.4 Database

Java offers a natural basis for the building of OO databases because of features such as serialization and reflection. There are already several promising OO databases products or prototypes, which may eventually satisfy HEP needs. This area needs further study.

7.5 Analysis and Graphics

Most of the mature Analysis packages such as JAS, WIRED, Atlantis and GraXML, are already written in Java or are in the process of migration. These programs are able to cooperate or stand alone. It seems very natural to use Java as the primary language for analysis.

7.6 User Interface

Java is recognised as having been a success for the GUI. Swing is very easy to use directly and there are many tools (in Java of course) which allow you to build your GUI graphically. There used to be a problem with speed of the Java graphics, this problem has been, however, already successfully solved except that Swing is still a bit slow. There is also the possibility to use Java via a command-line interface which makes it a good candidate as a scripting language (or as the interface to other scripting languages).

Java has been developed from the beginning as the language for building network-wide distributed client-server applications. Many of its features are directly aimed at the development of the application running transparently over the network (security, platform independence, remote message invocation, applets, servlets, mobile agents). If any language fits into the Grid environment, then it's Java. It is also worth noting that the Globus toolkit which many people are considering as the basis for Grid middleware is in ANSI C with a few Java tools around it. They avoided C++ for reasons of portability.

8. Recommendations

In general, we can say that there are no serious obstacles to introduce Java as the main implementation language for the HEP software development. All frequently claimed Java problems are either not problems at all, or they are already solved. At the same time many of the Java advantages will have clear positive impact on the functionality, performance and maintainability of our software. Also the number of existing standard and mature Java applications and class libraries will make the future software development for HEP much faster, safer and easier.

In a mixed Java-C++ environment, one can't fully exploit all the Java advantages. So it would be preferable to replace C++ by Java to benefit from OO without fighting the implementation language. Common experience shows, that once programmers start using Java, they won't go back. It appears that the time is right to start the move towards Java and we should not delay further.

8.1 Architecture

The ATLAS framework should be extended to allow transparent inter-operation of Java and C++ Algorithms with transparent access to Services in either language. Efficient access to (transient) data storage should be available to both languages.

The ATLAS coding conventions should be amended to discourage C++ features which may create problems for Java<-->C++ inter-operation and migration to Java. The most important is using non-abstract multiple inheritance and generic programing using templates.

8.2 ATLAS Support

Java should be fully supported in ATLAS. In particular, all standard tools including release tools and CASE tools, should be able to handle packages fully written in Java and some packages written in Java and others in C++.

8.3 CERN Support

ATLAS should request full support for Java from CERN IT. This includes installation and maintenance of JVMs and other needed Java libraries and applications as well as active development and support of key HEP Java packages, such as WIRED, JAS and Colt.

ATLAS should request a detailed study of the Java solutions and interoperability for software products developed or supported by CERN/IT. This concerns mostly LHC++, RD45 and GEANT4.

9. Acknowledgements

In this note, text from Documentation, Web Pages and Papers of the following authors has been used:

Mark Donszelmann
Wolfgang Hoschek
Tony Johnson

A. Appendix - Some useful Java products

This is just a short list of some Java products (potentially) useful for the software development in HEP. Description are mostly taken directly from the package description. The list is by no means exhaustive.

A.1. HEP

There are already several mature HEP products covering mostly the Domains of data analysis and basic computing infrastructure. Very often, those products are of higher standard than corresponding packages in Fortran/C/C++.

The Colt Distribution This provides an infrastructure for scalable scientific and technical computing in Java. It consists of several free Java libraries which have been repackaged for convenience along for user convenience bundled under one single uniform umbrella. Some of its man features are listed below:

Templated Lists and Maps.	Dynamically resizing lists holding objects or primitive data types. Operations on primitive arrays, algorithms on Colt lists and JAL algorithms can freely be mixed at zero copy overhead. Automatically growing and shrinking maps holding objects or primitive data types. Space efficient high performance BitVectors and BitMatrices.
Templated Multi-dimensional matrices	Dense and sparse fixed sized 1,2, 3 and n-dimensional matrices holding objects or primitive data types.
Linear Algebra	Standard matrix operations and decompositions. LU, QR, Cholesky, Eigenvalue, Singular value.
Histogramming.	Compact, extensible, modular and high performance histogramming functionality.
Mathematics	Tools for basic and advanced mathematics: Arithmetics and Algebra, Polynomials and Chebyshev series, Bessel and Airy functions, Constants and Units, Trigonometric functions, etc.
Statistics	Tools for basic and advanced statistics: Estimators, Gamma functions, Beta functions, Probabilities, Special integrals, etc.
Random Numbers and Random Sampling.	Strong yet quick. Partly a port of CLHEP.
JAL	A partial Standard Template Library port by Silicon Graphics.
util.concurrent	Efficient utility classes commonly encountered in parallel & concurrent programming.

See: http://nicewww.cern.ch/~hoschek/colt/index.htm

JAS Java Analysis Studio (JAS) is a graphical application for analysis of high-energy physics data. The application is independent of any particular data format, so that it can be used to analyze data from any experiment. The application features a rich graphical user interface (GUI) aimed at making the program easy to learn and use, but which at the same time allows the user to perform arbitrarily complex data analysis tasks by writing analysis modules in Java using the built in editor/compiler. The application can be used either as a stand-alone application, or as a client for a remote Java Data Server. The client-server mechanism is targeted particularly at allowing remote users to access large data samples stored on a central data center in a natural and efficient way. JAS is written entirely in Java and will run on any platform with a Java virtual machine..

Home	http://www-sldnt.slac.stanford.edu/jas/
In ATLAS	http://atlas.web.cern.ch/Atlas/GROUPS/GRAPHICS/Texts/Documentation/JAS/

that can be used across the network. Originally WIRED was developed to better understand the applicability of the Java technology in HEP. Now, it has grown to be a framework in use and under development in several experiments, both inside and outside CERN (ATLAS, CHORUS, DELPHI, LHCb, BaBar, D0 and ZEUS). To guarantee portability across all platforms WIRED uses the Java language and its Swing user interface component set. The graphical user interface allows for multiple views and for multiple controls acting on those views. A detector tree control is available to toggle the visibility of parts of the events and detector geometry. XML (Extensible Markup Language), RMI (Remote Method Invocation) and CORBA loaders can be used to load event data as well as geometry data, and to connect to FORTRAN, C, C++ and Java reconstruction programs. A special Java interpreter allows physicists to write small scripts to interact with their data and its display.

Home	http://wired.cern.ch/
In ATLAS	http://atlas.web.cern.ch/Atlas/GROUPS/GRAPHICS/Texts/Documentation/WIRED/

JACO Java Access for C++ Objects. Package being developed mostly by JAS and WIRED developers from requirements taken (among others) from ATLAS.

FreeHEP The goal of the FreeHEP library is to encourage the sharing and reuse of Java code in High Energy Physics. Although some of the code is fairly specific to HEP, other code is more generic and could be used by anyone. To maximize reuse the dependencies between various packages in the FreeHEP library are kept to a minimum, so you can use which ever parts interest you. The FreeHEP Java library is an "Open Source" library distributed under the terms of the LGPL.

See: http://java.freehep.org/lib/freehep/doc/web/index.html

A.2. non HEP

The amount of existing high-quality java code is enormous.

Xerces/XML4J XML Parser for Java is a validating XML parser written in 100% pure Java. The package contains classes and methods for parsing, generating, manipulating, and validating XML documents.

See: http://xml.apache.org/xerces-j/index.html

LOG4J With log4j it is possible to control logging at runtime without modifying the compiled code. The log4j package is designed so that logging calls, which have been disabled by the control file, do not incur a heavy performance cost. This is a small, but well designed package.

See: http://www.log4j.org/

BeanShell BeanShell is a small, free, embedable, Java source interpreter with object scripting language features, written in Java. BeanShell executes standard Java statements and expressions, in addition to obvious scripting commands and syntax. BeanShell supports scripted objects as simple method closures like those in Perl and JavaScript□ .

You can use BeanShell interactively for Java experimentation and debugging or as a simple scripting engine for you applications. You can call BeanShell from your Java applications to execute Java code dynamically at run-time or to provide scripting extensibility for your applications. Alternatively, you can call your Java applications and objects from BeanShell; working with Java objects and APIs dynamically. Since BeanShell is written in Java and runs in the same space as your application, you can freely pass references to "real live" objects into scripts and return them as results.

See: http://www.beanshell.org/intro.html

DynamicJava DynamicJava is a Java source interpreter with a functionality similar to the BeanShell. It is free and distributed with the source code.

See: http://www.inria.fr/koala/djava/

Ozone Ozone is a fully featured, object-oriented database management system completely

ozone project is to evolve a technology that allows developers to build pure object-oriented, pure Java database applications. Just program your Java objects and let them run in a transaction based, database environment.

See: http://www.ozone-db.org/

Java3D The Java 3D API is a set of classes for writing three-dimensional graphics applications. It gives developers high level constructs for creating and manipulating 3D geometry and for constructing the structures used in rendering that geometry.

See: http://java.sun.com/products/java-media/3D/

X3D X3D (Extensible 3D) is an inter-operable set of lightweight, componentized 3D standards. It incorporates a number of component specifications allowing extremely lightweight applications to be deployed on a variety of platforms. Initial components include a lightweight 3D runtime engine with state-of-the-art rendering capabilities, a platform-independent 3D file format and advanced XML integration.

See: http://www.web3d.org/x3d.html

JDOM JDOM offers an easy to use Java API to manipulate XML documents. While JDOM inter-operates well with existing standards such as the Simple API for XML (SAX) and the Document Object Model (DOM), it is not an abstraction layer or enhancement to those APIs. Rather, it seeks to provide a robust, light-weight means of reading and writing XML data.

See: http://www.jdom.org/

JBuilder JBuilder is one of many IDEs for developing Java programs. It is quick to learn, and is free in its basic form and offers very good debugging facilities and a GUI builder.

See: http://www.inprise.com/jbuilder/